

CS241 Operating Systems

CPU Scheduling (4)

Klara Nahrstedt

Lecture 13

2/17/2006

Content

- Signals
- Introduction to Timers
 - POSIX:TMR
- Summary

Administrative Notes

- MP2 on scheduling is running

Manipulating Signal Masks and Signal Sets

- A process can temporarily prevent a signal from being delivered by **blocking it**
- Process **signal mask** gives the set of currently blocked signals
- Signal mask is of type `sigset_t`
- There is **difference** between blocking a signal and ignoring a signal
- Protocol for blocking a signal
 - Process blocks a signal by modifying its signal mask (use `sigprocmask`)
 - OS does not deliver signal until process unblocks the signal
- Protocol for ignoring a signal
 - When a signal is delivered to process, process throws it away
 - Process sets a signal to be ignored by calling `sigaction` with handler of `SIG_IGN`

Manipulation Operations of Signal Sets

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

sigemptyset initializes the set to contain no signals
sigfillset puts all signals in the set
sigaddset adds one signal to the set
sigdelset removes one signal from the set
sigismember tests to see if a signal is in the set

Example: initialize a signal set:

- Initialize signal set **twosigs** to contain exactly the two Signals **SIGINT** and **SIGQUIT**

```
if ((sigemptyset(&twosigs) == -1) ||
    (sigaddset(&twosigs, SIGINT) == -1) ||
    (sigaddset(&twosigs, SIGQUIT) == -1))
    perror("Failed to set up signal mask");
```

Signaling and Blocking

- The process signal mask is a set of signals that are currently blocked.
- A blocked signal remains pending after it is generated until the signal is unblocked.
- The process signal mask is modified with the **sigprocmask** system call.

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t *restrict set,  
               sigset_t *restrict oset);
```

The **how** parameters can be:

SIG_BLOCK /* add a collection of signals to those currently blocked*/

SIG_UNBLOCK /* delete a collection of signals from currently blocked*/

SIG_SETMASK /*set the collection of signals being blocked to the specified set*/

Either set or oset may be NULL

Add SIGINT to the set of blocked signals

```
sigset_t newsigset;
```

```
if ((sigemptyset(&newsigset) == -1) ||  
    (sigaddset(&newsigset, SIGINT) == -1))  
    perror("Failed to initialize the signal set");  
else if (sigprocmask(SIG_BLOCK, &newsigset, NULL)  
        == -1)  
    perror("Failed to block SIGINT");
```

If SIGINT is already blocked, the call to **sigprocmask** has no effect.

Catching and Ignoring Signals: **sigaction**

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
              struct sigaction *oact);

struct sigaction {
    void (*sa_handler)(int); /* SIG_DFL, SIG_IGN or pointer to
                               function */
    sigset_t sa_mask; /* additional signals to be blocked during
                       execution of handler */
    int sa_flags; /* special flags and options */
    void (*sa_sigaction) (int, siginfo_t *, void *); /* realtime
handler */
};
```

Either act or oact may be NULL. If the SA_SIGINFO flag of the sa_flags field is clear, sa_handler specifies the action to be taken. void (*sa_handler)() means a pointer to a function that has no return value.

Set up a signal handler for SIGINT

```
void catchctrlc(int signo) {
    char handmsg[] = "I found Ctrl-C\n";
    int msglen = sizeof(handmsg);

    write(STDERR_FILENO, handmsg, msglen);
}

... struct sigaction act;
act.sa_handler = catchctrlc;
act.sa_flags = 0;
if ((sigemptyset(&act.sa_mask) == -1) ||
    (sigaction(SIGINT, &act, NULL) == -1))
    perror("Failed to set SIGINT to handle Ctrl-C");
```

Note: write is async-signal safe – meaning it can be called inside a signal handler. Not so for printf or strlen.

A correct way to wait for a signal

1. `static volatile sig_atomic_t sigreceived = 0;`
- 2.
3. `sigset_t maskblocked, maskold, maskunblocked;`
4. `int signum = SIGUSR1;`
5. `sigprocmask(SIG_SETMASK, NULL, &maskblocked);`
6. `sigprocmask(SIG_SETMASK, NULL, &maskunblocked);`
7. `sigaddset(&maskblocked, signum);`
8. `sigdelset(&maskunblocked, signum);`
9. `sigprocmask(SIG_BLOCK, &maskblocked, &maskold);`
10. `while(sigreceived == 0)`
11. `sigsuspend(&maskunblocked);`
12. `sigprocmask(SIG_SETMASK, &maskold, NULL);`

Note. Assume signal handler **Sigreceived** returns a 1 for **signum**.
Error codes omitted for clarity.

Waiting for Signals

sigwait

- Block all signals
- Put the signals you want to wait for in a **sigset_t**
- call **sigwait**
- **sigwait** blocks the process until at least one of these signals is pending.
- It removes one of the pending signals and gives you the corresponding signal number in the second parameter..
- Do what you want: no signal handler needed.
- It returns 0 on success and -1 on error with errno set.

```
#include <signal.h>  
int sigwait(const sigset_t *restrict sigmask, int *restrict signo);
```

Signal Handling and Threads ch13.5

p473-475

type	Delivery action
asynchronous	Delivered to some thread that it has unblocked
synchronous	Delivered to thread that caused it
directed	Delivered to the identified thread

Directing a Signal to a Particular Thread

```
#include <signal.h>  
#include <pthread.h>
```

```
int pthread_kill(pthread_t thread, int sig);
```

If successful `pthread_kill` returns a 0.

```
If (pthread_kill(pthread_self(), SIGKILL))  
    fprintf(stderr, "failed to commite suicide"\n");
```

Masking Signals for Threads

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int how, const sigset_t
*restrict set,
                    sigset_t *restrict pset);
```

- Each thread has its own signal mask for process wide signals – only use **sigprocmask** for initialization before creating threads
- Can now use signals for thread specific handlers

Dedicating threads for signal handling

- Dedicate particular threads to handle signals
- Main thread blocks all signals before creating any child threads
- Signal mask is inherited by children
- Thread dedicated to signal handling executes `sigwait` on specific signals to be handled or uses `pthread_sigmask` to unblock signal

POSIX Times

- POSIX specifies time in seconds since the Epoch
- Epoch is defined as 00:00 (midnight), January 1, 1970, Coordinated Universal Time (UTC)
- Two POSIX Extensions
 - POSIX:TRM (used also in our MP)
 - POSIX:XSI

Expressing, Displaying Time

- POSIX base standard supports only time resolution of seconds
- Expression of time as `time_t` type

```
#include <time.h>
```

```
time_t time(time_t *tloc)
```

- Displaying current time

```
#include <stdio.h>
```

```
#include <time.h>
```

```
int main(void) {
```

```
    time_t tcurrent;
```

```
    tcurrent = time(NULL);
```

```
    printf("the current time is %s", ctime(&tcurrent));
```

```
    return 0;
```

```
}
```

POSIX:TRM Extension (1)

- A *clock* is a counter – increments at fixed intervals and represents the *clock resolution*
- POSIX:TRM represents clocks as variables of type **clockid_t**
- All implementations of POSIX clocks must support systemwide clock with **clockid_t** value **CLOCK_REALTIME** (corresponds to system realtime clock)
- Finer granularity clock structure

struct timespec has the following parameters:

time_t tv_sec /*seconds*/

long tv_nsec /*nanoseconds*/

POSIX:TRM Extension (2)

- Important Clock Functions
 - Set the clock time – **clock_settime**
 - Retrieve the clock time – **clock_gettime**
 - Determine the clock resolution – **clock_getres**

```
#include <time.h>
```

```
int clock_getres(clockid_t clock_id, struct  
timespec *res);
```

```
int clock_gettime(clockid_t clock_id, struct  
timespec *tp);
```

```
int clock_settime(clockid_t clock_id, const struct  
timespec *tp);
```

Time Measurements

(Get Processing Time of Executing Function)

```
#include <stdio.h>
#include <time.h>
#define MILLION 1000000L

void function_to_time(void);
int main(void) {
    long timedif;
    struct timespec tpend, tpstart;
    if (clock_gettime(CLOCK_REALTIME, &tpstart) == -1) {
        perror("Failed to get starting time");
        return 1; }
    function_to_time();
    if (clock_gettime(CLOCK_REALTIME, &tpend) == -1) {
        perror("Failed to get ending time");
        return 1; }
    timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) + (tpend.tv_nsec - tpstart.tv_nsec)/1000;
    printf("The function_to_time took %ld microseconds \n", timedif);
    return 0;
}

function_to_time(void) {
    int x;
    int i;
    x=1;
    for (i=0;i<100; i++) { x++;}
}
```

Summary

- Signals
- Timers – POSIX TRM